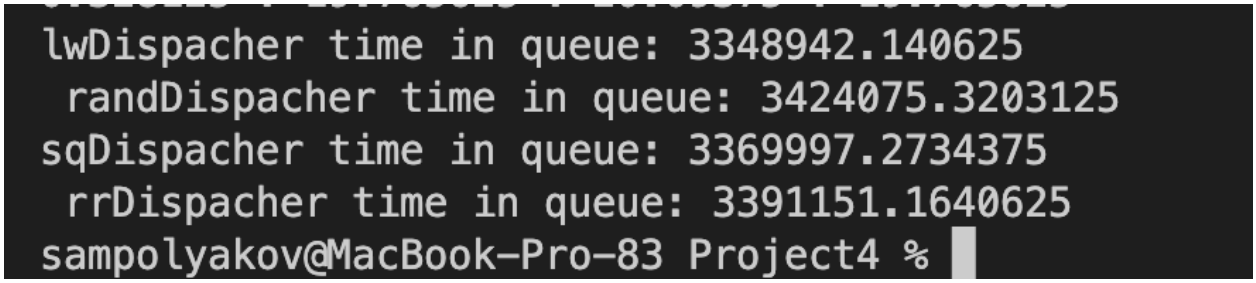
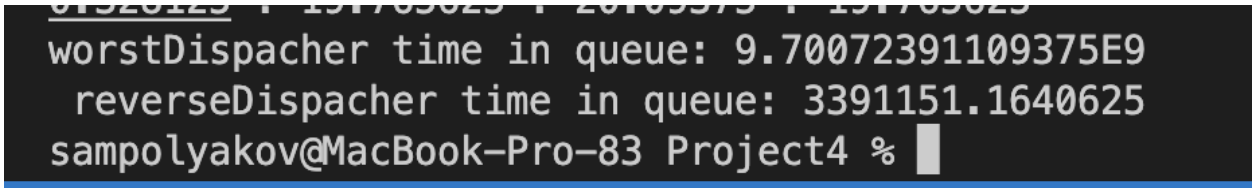
Sam Polyakov
Project 4
3/5/2023
CS231B

Modeling a Server Farm

Abstract:
This week, we created a project that modeled a server farm. We used java classes and objects, array lists, arrays, methods, loops, abstract classes, inheritance, Linked Lists, iterators, and Queues to build all of the classes. In the end, the program created 6 “Job Dispatchers” which assigned jobs to servers in different orders, based on the instructions they were given. After running the program, it would print out the average time in queue for a job based on how the Job Dispatchers assigned jobs to each server. For my extension, I added the ability for the user to input how many servers should be created using command-line arguments, and I created 2 additional job dispatchers with different functionalities.

Results:



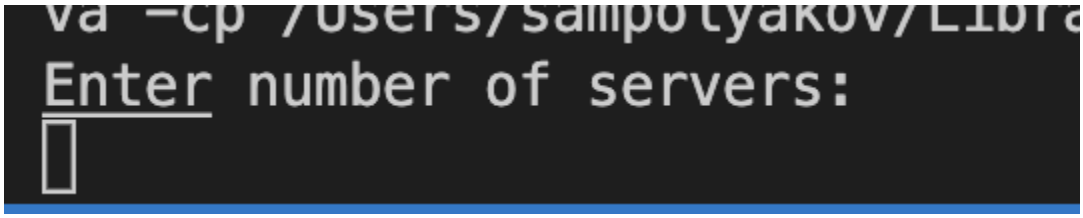


Overall, this simulation of a server farm produced the outcome I was expecting, with LeastWorkDispacher being the most efficient and RandomDispatcher being the least efficient (other than my WorstDispatcher extension). I predicted this because with LeastWorkDispatcher, the longest queue should be shorter than with the other dispatchers, as they would all be more or less equal. RandomDispatcher, on the other

hand, could have queues of any size where one queue could be hours long while the others were very short. I also predicted that ShortestQueueDispatcher would be the second best. I predicted this because even though it was possible for one Queue to be longer than another simply because it happened to have more jobs with long processing times, that would have less and less of an effect over time. Finally, I predicted that roundRobinDispacher would be the third best as the queues size would not be as even as in ShortestQueueDispacher over time, but it would be better than in RandomDispacher. Finally, for my extension, I added ReverseRoundRobinDispacher and WorstDispacher. I predicted that ReverseRoundRobinDispatcher would perform just as well as RoundRobinDispatcher because it was effectively doing the same thing, just in reverse. I also predicted that WorstDispatcher would be by far the slowest, as all of its jobs got added into one queue.

Extension:

For the first part of my extension, I added command-line arguments when running the exploration. Now, the user can input how many servers they want there to be. I did this by importing java.util.Scanner and using scanner.nextInt to get the input number.



For the second and third parts of my extension, I created 2 additional dispatchers. The first one, WorstDispacher, was created to see how long the longest possible queue

would be. To do this, I sent all of the jobs to 1 server instead of spreading them out evenly among all the servers, like I did with the other dispatchers. Unsurprisingly, the WorstDispatcher was significantly less efficient than the others, with it taking almost 3000 times longer than RandomDispacher, which was previously the worst. I also created a ReverseRoundRobinDispacher which did the same thing as RoundRobin, but in reverse. I did this because I wanted to see what effect, if any, it had on the runtime of using this strategy. In the end, it ended up having the exact same runtime as RoundRobin, which makes sense to me.

```
worstDispacher time in queue: 9.70072391109375E9
 reverseDispacher time in queue: 3391151.1640625
sampolyakov@MacBook-Pro-83 Project4 %
```

```java
public class ReverseRoundRobinDispatcher extends JobDispatcher{
  // dispaches jobs to servers in a reverse round-robin order

    private LinkedList<Server> serversAvailable;
    int counter;

    public ReverseRoundRobinDispatcher(int k) {
        // specifies number of servers
        super(k);
        counter = k-1;
    }

    public Server pickServer(Job j) {
        // returns Servers in a reverse round-robin process
        serversAvailable = super.serversMaintained;

        if(counter < 0){
            counter = serversAvailable.size()-1;
        }
        counter--;

        return serversAvailable.get(counter+1);
    }

}
```

```java
import java.util.Random;

public class WorstDispacher extends JobDispatcher{
  // creates worst possible dispacher

    protected LinkedList<Server> serversAvailable;
    Random rand = new Random() ;

    public WorstDispacher(int k) {
        // specifies number of servers
        super(k);
    }

    public Server pickServer(Job j) {
        // puts all work in 1 server
        serversAvailable = super.serversMaintained;
        return serversAvailable.get(0);
    }

}
```

Reflection:

This week, we focused on Queues. Queues are used for “first in, first out”, like putting a Job into a servers queue and taking it out once the server has finished processing that

job. The peek, poll, and offer methods are all very efficient and have a constant worst-case runtime, because you only need to know the first and last elements of the queue for these to work. In our server farm, the most efficient method was LeastWorkDispacher with a constant run time. The next fastest, ShortestQueueDispacher also always had a constant runtime. After this, they all had variable runtimes. This shows how much more efficient having a constant runtime is.

References:

This week, I worked with Dave Boku and got help from Max Bender at office hours.